Backpropagation

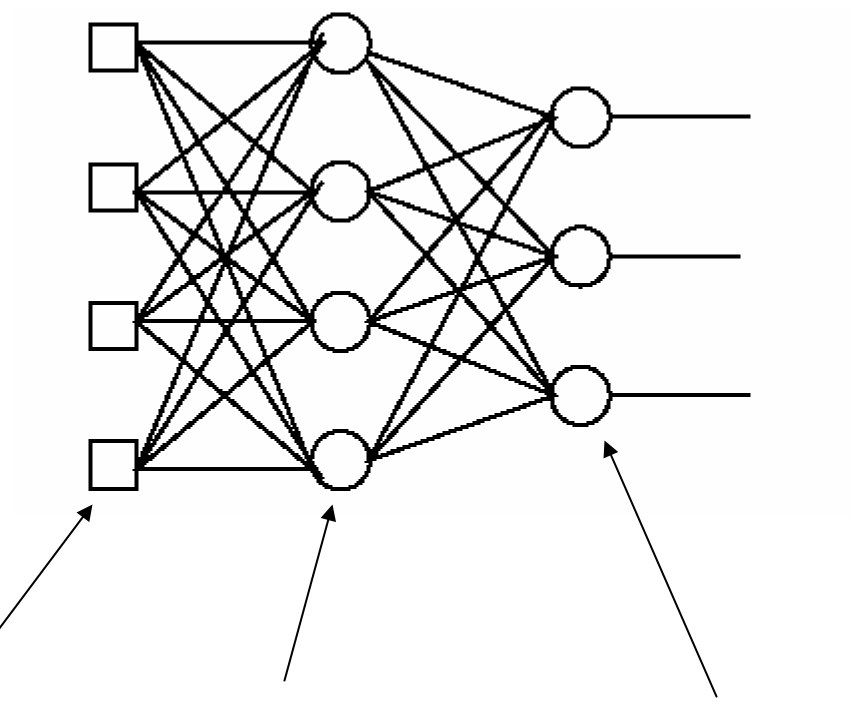
Using multilayer networks it is possible to represent the XOR function.
In fact, it can be shown that *any* continuous function can be approximated with arbitrary accuracy using a 2-layer network. A 3-layer network can approximate any non-continuous function with arbitrary accuracy.

However, in order for a multilayer network to be useful, a non-linear activation function (sigmoid) must be used. In fact, if the activation function is linear a multilayer network is equivalent with a one-layer network:

$$\mathrm{y}^H = \mathrm{w}^{I \to H}\mathrm{x}, \qquad \mathrm{y}^O = \mathrm{w}^{H \to O}\mathrm{y}^H = \mathrm{w}^{H \to O}\mathrm{w}^{I \to H}\mathrm{x} \equiv \mathrm{w}\mathrm{x},$$

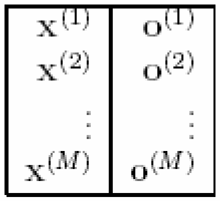
where

$$\mathrm{w} = \mathrm{w}^{H \to O}\mathrm{w}^{I \to H}.$$



Input layer ($x = y^I_i$) Hidden layer (y^H_i) Output layer (y^O_i)

Assume that a training set, i.e. a set of *M* input-output pairs, has been defined as shown in the table:



$\mathbf{x}^{(1)}$	$\mathbf{o}^{(1)}$
$\mathbf{x}^{(2)}$	$\mathbf{o}^{(2)}$
$\vdots$	$\vdots$
$\mathbf{x}^{(M)}$	$\mathbf{o}^{(M)}$

For any input vector $\boldsymbol{X} = \boldsymbol{X}^{(m)}$, the components $e_l = e_l^{(m)}$ of the error $\boldsymbol{e}^{(m)}$ can be computed as

$$e_l = o_l - y_l^O ,$$

where $y_l^{(O)}$ is the output from neuron *l*.

For output neurons it is thus easy to compute the error. However, for the neurons in the hidden layer, the error cannot be computed directly. We are therefore faced with a **credit assignment problem**: how should neurons in the hidden layer be rewarded (or punished) for their performance?

First, however, let us consider the output neurons. The total error for the input vector $\boldsymbol{X}^{(m)}$ can be defined as

$$\mathcal{E}^{(m)} = \frac{1}{2}\sum_{l=1}^{n^{(O)}} (e_l^{(m)})^2.$$

In backpropagation, it is common to use stochastic gradient descent. Thus, the input-output pairs $\{\boldsymbol{X}^{(m)}, \boldsymbol{O}^{(m)}\}$ are considered one at a time, and we can therefore drop the superscript (m), in order to reduce the number of indices. Thus we set

$$\mathcal{E} \equiv \mathcal{E}^{(m)}$$

The derivation of the backpropagation rule proceeds in approximately the same way as the derivation of the delta rule: the basic idea is to find the direction of steepest descent (i.e. the negative gradient) and then follow this direction towards smaller training errors:

$$\Delta w_{ij}^{H\to O} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^{H\to O}} \qquad \Delta w_{ij}^{I\to H} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^{I\to H}}.$$

where $w_{ij}^{H\to O}$ is the weight connection neuron *j* in the hidden layer with neuron *i* in the output layer, and $w_{ij}^{I\to H}$ is the weight connection input element *j* with neuron *i* in the hidden layer.

The derivative for of the error with respect to the weight connecting the hidden layer to the output layer can easily be computed as

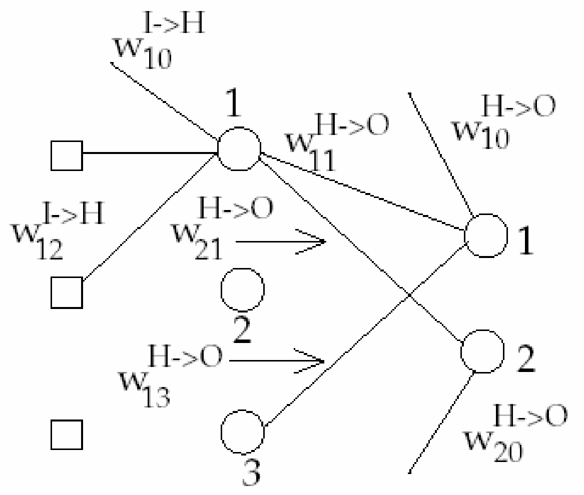
$$
\begin{aligned}
\frac{\partial \mathcal{E}}{\partial w_{ij}^{H\to O}} &= \frac{\partial}{\partial w_{ij}^{H\to O}}\left(\frac{1}{2}\sum_{l=1}^{n^{(O)}} e_l^2\right) = e_i \frac{\partial e_i}{\partial w_{ij}^{H\to O}} = \\
&= e_i \frac{\partial}{\partial w_{ij}^{H\to O}}\left(o_i - \sigma\left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\to O} y_s^H\right)\right) = \\
&= -e_i \sigma'\left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\to O} y_s^H\right) \frac{\partial\left(\sum_{s=0}^{n^{(H)}} w_{is}^{H\to O} y_s^H\right)}{\partial w_{ij}^{H\to O}} = \\
&= -e_i \sigma' y_j^H,
\end{aligned}
$$

The modifications of the weights connecting the hidden layer to the output layer can now be computed as

$$\Delta w_{ij}^{H \to O} = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^{H \to O}} = \eta \delta_i y_j^H ,$$

where $\delta_i = e_i \sigma'$ is the *local gradient*. Except for the σ', the expression for the weight modification is the same as in the derivation of the delta rule (where σ' was equal to 1).

Consider now the neurons in the hidden layer. These neurons are connected to *all* neurons in the output layer, and will therefore affect the error for all those neurons.



The partial derivative with respect to the weights connecting the input elements to the hidden layer can be obtained (using the chain rule) as

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{I\to H}} = \frac{\partial \mathcal{E}}{\partial y_i^H}\frac{\partial y_i^H}{\partial w_{ij}^{I\to H}} = \frac{\partial \mathcal{E}}{\partial y_i^H}\frac{\partial}{\partial w_{ij}^{I\to H}}\sigma\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) =$$

$$= \frac{\partial \mathcal{E}}{\partial y_i^H}\sigma'\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) y_j^I,$$

Continuing this calculation, we find

$$\begin{aligned}
\frac{\partial \mathcal{E}}{\partial y_i^H} &= \frac{\partial}{\partial y_i^H}\left(\frac{1}{2}\sum_{l=1}^{n^{(O)}} e_l^2\right) = \sum_{l=1}^{n^{(O)}} e_l \frac{\partial e_l}{\partial y_i^H} = \sum_{l=1}^{n^{(O)}} e_l \frac{\partial (o_l - y_l^O)}{\partial y_i^H} = \\
&= -\sum_{l=1}^{n^{(O)}} e_l \frac{\partial}{\partial y_i^H}\sigma\left(\sum_{s=0}^{n^{(H)}} w_{ls}^{H\to O} y_s^H\right) =
\end{aligned}$$

$$\begin{aligned}
&= -\sum_{l=1}^{n^{(O)}} e_l \sigma'\left(\sum_{s=0}^{n^{(H)}} w_{ls}^{H\to O} y_s^H\right) w_{li}^{H\to O} = \\
&= -\sum_{l=1}^{n^{(O)}} \delta_l w_{li}^{H\to O},
\end{aligned}$$

Thus, the weight modification can be written

$$\Delta w_{ij}^{I\to H} = \eta \kappa_i y_j^I,$$

where

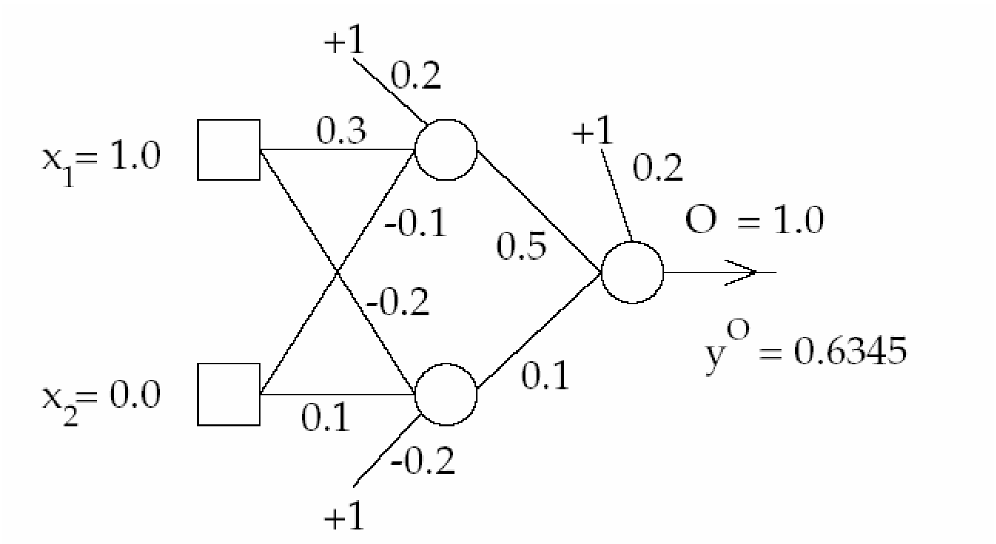
$$\kappa_i = \sigma'\left(\sum_{p=0}^{n^{(I)}} w_{ip}^{I\to H} y_p^I\right) \sum_{l=1}^{n^{(O)}} \delta_l w_{li}^{H\to O}.$$

This completes the derivation of the backpropagation algorithm. The algorithm derives its name from the fact that the errors are propagated backwards through the network. Thus in order to compute the κ, one must first know the δ etc.

Note that the modifications Δw_{ij} are computed for *all* weights in the network before *any* modifications are actually applied. Thus, in the computation of the modifications for the weights connecting the input layer to the hidden layer, the old values are used for the weights $w_{ij}^{H \to O}$.

Example

The backpropagation algorithm can be illustrated by means of an example. Consider the network shown in the figure below:



The aim of the training is to arrive at a set of weights for which the input x_1=1.0, x_2=0.0 gives the output o=1.0.

The first step is to compute the output from the network, starting with the output from the hidden layer:

$$\begin{aligned}
y_1^H &= \sigma\left(\sum_{p=0}^{2} w_{1p}^{I\to H} y_p^I\right) = \sigma(0.2 \times 1.0 + 0.3 \times 1.0 - 0.1 \times 0) = \\
&= \sigma(0.5) = \frac{1}{1+\mathrm{e}^{-0.5}} = 0.6225, \\
y_2^H &= \sigma\left(\sum_{p=0}^{2} w_{2p}^{I\to H} y_p^I\right) = \sigma(-0.2 \times 1.0 - 0.2 \times 1.0 + 0.1 \times 0) = \\
&= \sigma(-0.4) = \frac{1}{1+\mathrm{e}^{0.4}} = 0.4013.
\end{aligned}$$

$$\begin{aligned}
y_1^O &= \sigma\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^H\right) = \sigma(0.2 \times 1.0 + 0.5 \times 0.6225 + 0.1 \times 0.4013) = \\
&= \sigma(0.5514) = 0.6345.
\end{aligned}$$

The next step is to compute the error and the local gradient: (only one such term here, since the output layer consists of a single neuron)

$$e_1 = o_1 - y_1^O = 1 - 0.6345 = 0.3655.$$

$$\sigma'\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^H\right) = \sigma'(0.5514) = 0.2319.$$

$$\delta_1 = e_1 \sigma'\left(\sum_{s=0}^{2} w_{1s}^{H\to O} y_s^I\right) = 0.3655 \times 0.2319 = 0.0848.$$

Now the modifications of the weights connecting the hidden layer to the output layer can be computed:

$$\Delta w_{10}^{H\to O} = \eta \delta_1 y_0^H = 1 \times 0.0848 \times 1 = 0.0848.$$

$$\Delta w_{11}^{H\to O} = \eta \delta_1 y_1^H = 1 \times 0.0848 \times 0.6225 = 0.0528,$$
$$\Delta w_{12}^{H\to O} = \eta \delta_1 y_2^H = 1 \times 0.0848 \times 0.4013 = 0.0340.$$

Proceeding with the hidden layer, we find

$$\begin{aligned}\kappa_1 &= \sigma'\left(\sum_{p=0}^{2} w_{1p}^{I\to H} y_p^I\right)\sum_{l=1}^{1}\delta_l w_{l1}^{H\to O} = \\ &= \sigma'(0.5)\delta_1 w_{11}^{H\to O} = 0.2350 \times 0.0848 \times 0.5 = 0.00996.\end{aligned}$$

The weight modifications for the first neuron in the hidden layer thus become

$$\Delta w_{10}^{I\to H} = \eta\kappa_1 y_0^I = 1 \times 0.00996 \times 1 = 0.00996.$$

$$\begin{aligned}\Delta w_{11}^{I\to H} &= 0.00996, \\ \Delta w_{12}^{I\to H} &= 0.\end{aligned}$$

Similarly, for the second neuron in the hidden layer the weight modifications are given by

$$\begin{aligned}
\Delta w_{20}^{I\to H} &= 0.00204,\\
\Delta w_{21}^{I\to H} &= 0.00204,\\
\Delta w_{22}^{I\to H} &= 0.
\end{aligned}$$

Updating all the weights, and again computing the output, we find (using, of course, the same input signals)

$$y_1^O = 0.6649.$$

Thus, the error has decreased from 0.3655 to 1-0.6649=0.3351.

Comments on backpropagation

Momentum: the convergence of the backpropagation algorithm can be improved through the introduction of a so called momentum term

$$\Delta w_{ij}^{H\rightarrow O}(t) = \epsilon \Delta w_{ij}^{H\rightarrow O}(t-1) + \eta \delta_i y_j^{(H)},$$

(and similarly for the weights connecting the input elements to the hidden layer). Thus, here the weight modification is not only dependent on the present input, but also on the previous weight modification.

Varying the learning rate: if the learning rate (η) is too large, there is a risk of oscillatory behavior for the training error *E*. On the other hand, if the learning rate is too small, convergence towards small training errors is slow.

Usually, the learning rate can be set rather high at the beginning of a run.

Two schemes for modifying the learning rate:

$$\eta(n) = \eta_0 \frac{1 + \frac{c}{\eta_0}\frac{n}{T}}{1 + \frac{c}{\eta_0}\frac{n}{T} + T\frac{n^2}{T^2}}.$$

$$\Delta\eta = \begin{cases} a & \text{if } \Delta E < 0 \text{ consistently;} \\ -b\eta & \text{if } \Delta E > 0 \text{ consistently;} \\ 0 & \text{otherwise,} \end{cases}$$

Applications of backpropagation

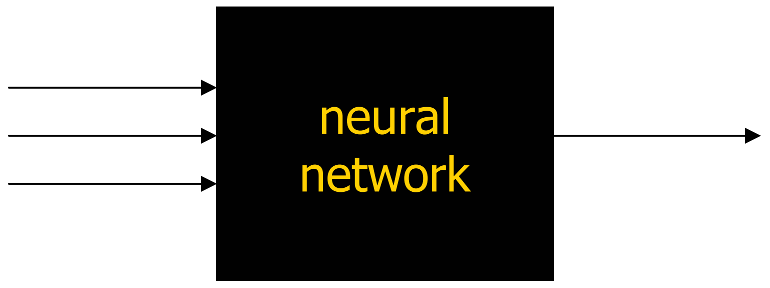
Backpropagation has many applications including

- System identification
- Function approximation
- Time series prediction
- Image recognition

Using backpropagation

"Neural networks are the second best solution to just about any problem"

Neural networks can, in principle, be applied to almost all approximation and prediction tasks. However, a neural network most often represents a black-box solution to a problem:



Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde

Thus, in cases where a proper physical model exists, it is often better to perform parameter-tuning within that model, rather than using a neural network.

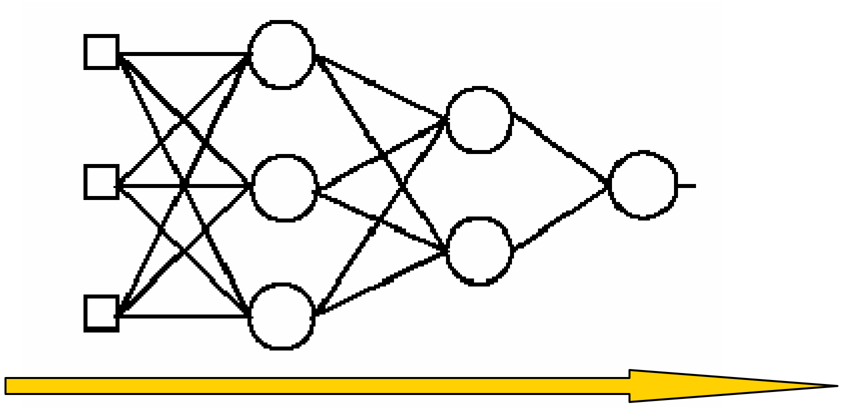
However, in many cases, no physical model is available (or, if a model is available, it may be too complex to use). In such cases, the use of a neural network is well motivated.

Neural network have additional advantages, such as (1) the ability to interpolate between data points, even in cases where the approximated function is non-linear, (2) **graceful degradation**, which is of particular importance in hardware implementations.

Function approximation

Feedforward neural networks (FFNN) are often used for function approximation.

Even though the training times (using backpropagation) may be long, the execution (i.e. computation of output) is almost instantaneous.



Rapid computation of output

Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde

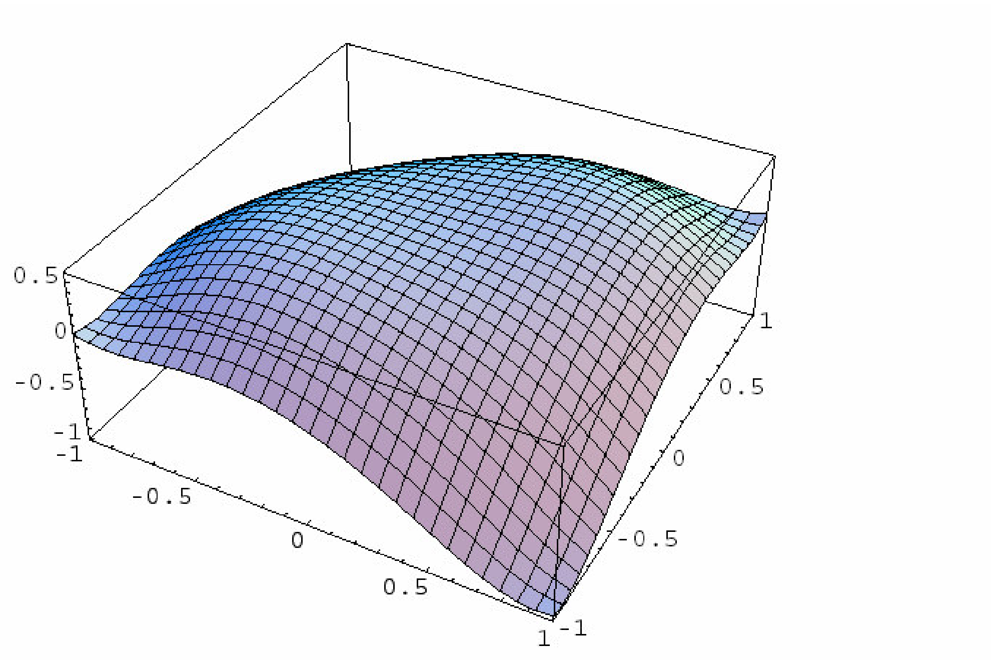
Function approximation: example

Consider the function

$$f(x, y) = \frac{1}{2}\left(\sin\sqrt{2}xy + \cos\sqrt{3}(x^2 + y^2)\right)$$

on the range x,y in [-1,1].

A plot of the function is shown in the figure on the right.



In order to approximate this function with a neural network, a training data set must be obtained by sampling the function. Here, the function was sampled at 21 equidistanct points along each axis, to generate a total of
21^2 = 441 training data points.

A validation set containing 400 points was generated by sampling points between those used in the training set.

Parts of the training data set and the validation data set are shown in the table on the right.

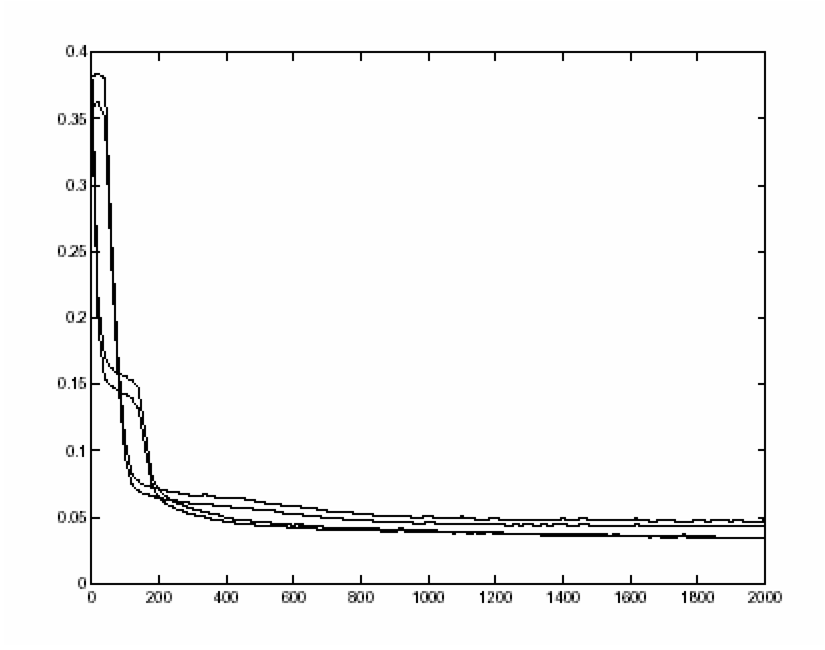
```
FunctionApproximation_Training
2 1
-1.0000 -1.0000 0.0197
-1.0000 -0.9000 -0.0220
-1.0000 -0.8000 -0.0250
-1.0000 -0.7000 -0.0054
-1.0000 -0.6000 0.0218
-1.0000 -0.5000 0.0449
-1.0000 -0.4000 0.0558
...
1.0000 0.8000 -0.0250
1.0000 0.9000 -0.0220
1.0000 1.0000 0.0197
```

Training data set

```
FunctionApproximation_Validation
2 1
-0.9500 -0.9500 -0.0215
-0.9500 -0.8500 -0.0188
-0.9500 -0.7500 0.0113
-0.9500 -0.6500 0.0520
-0.9500 -0.5500 0.0899
...
```

Validation data set

For the training (using the Backpropagation_v1.0 software, a 2-4-1 FFNN was used, with tanh(cz) as the activation function. A 2-10-1 FFNN was also tried.



The graph shows the training and validation error for both runs. The error for the network with 10 hidden neurons drops rapidly at first, then reaches a plateau, and finally continues to drop to slightly lower values than the error obtained for the network with 4 hidden neurons.

Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde

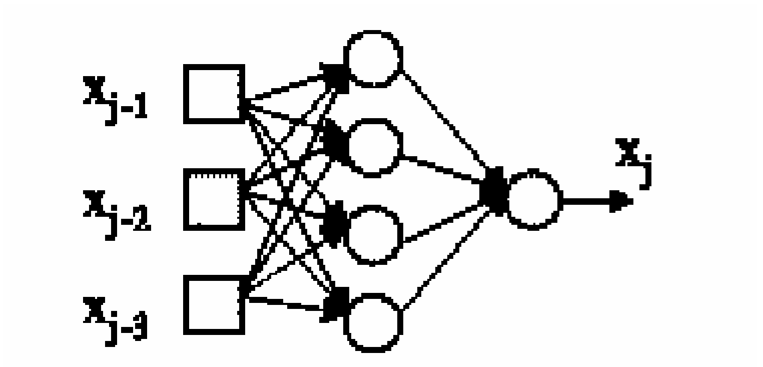
Time series prediction

Consider a series of values $x(i)$, $i = 1,2,\ldots,n$. **Time series prediction** is the problem of predicting $x(j + j_+)$, for all relevant values of j, given a set of earlier values $\{x(i - j_1), x(i - j_2),\ldots, x(i - j_N)\}$, where, commonly $j_+=0$, $j_1 = 1$, $j_2 = 2$ etc.

N is called the lookback.

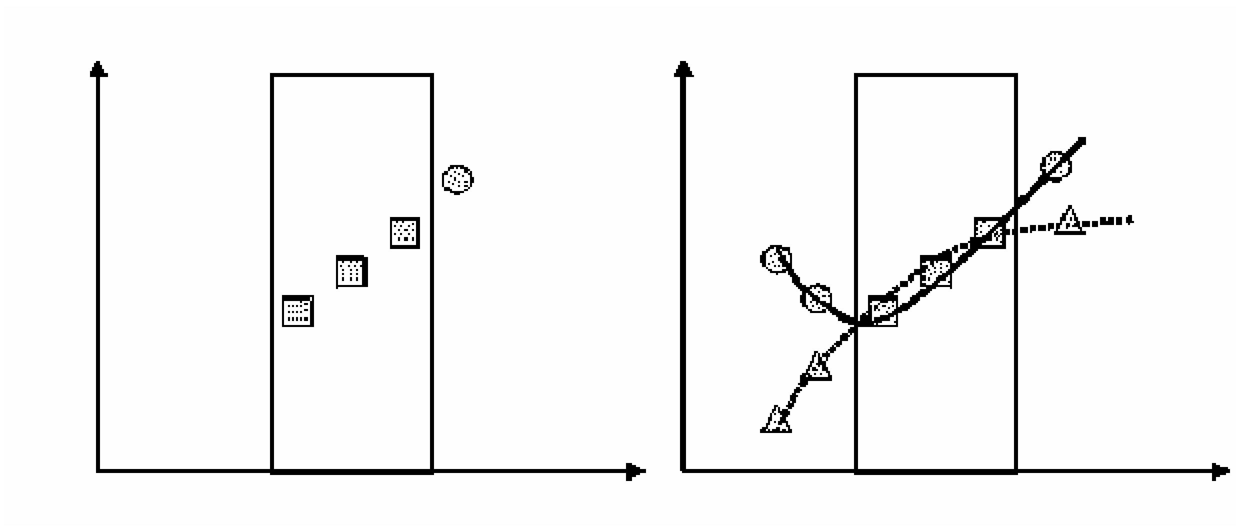
Common applications. Prediction of

- Financial time series
- Macroeconomic time series
- Medical and epidemiological time series
- Meteorological time series
- Earthquake data time series



An FFNN for time series prediction, with a lookback of $N = 3$.

A limitation of FFNN for time series prediction is their lack of dynamic short-term memory. FFNN have, of course, a long-term memory encoded in the connection weights obtained using (backpropagation) training. However, the response to any given input is always the same, as illustrated in the figure on the next slide


An illustration of the lack of dynamic short-term memory in FFNN.

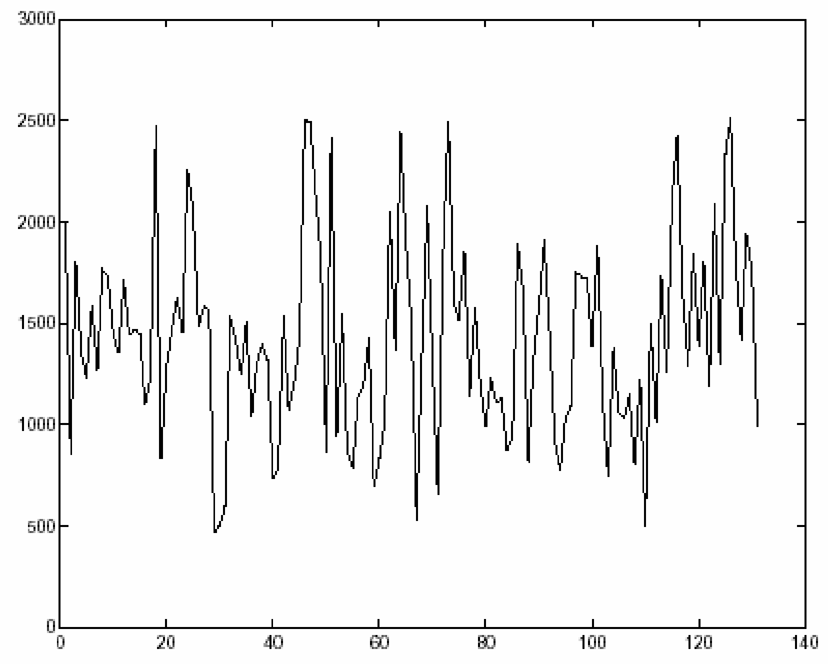
By contrast, recurrent neural networks (RNN) do have dynamic short-term memory, but are more difficult to train.

Example of time series prediction: Rainfall in Fortaleza, Brazil.

The figure shows the annual rain fall in Fortaleza, Brazil, from 1849 to 1979.

A conversion software was used for converting the raw data into a form suitable for use with FFNNs.

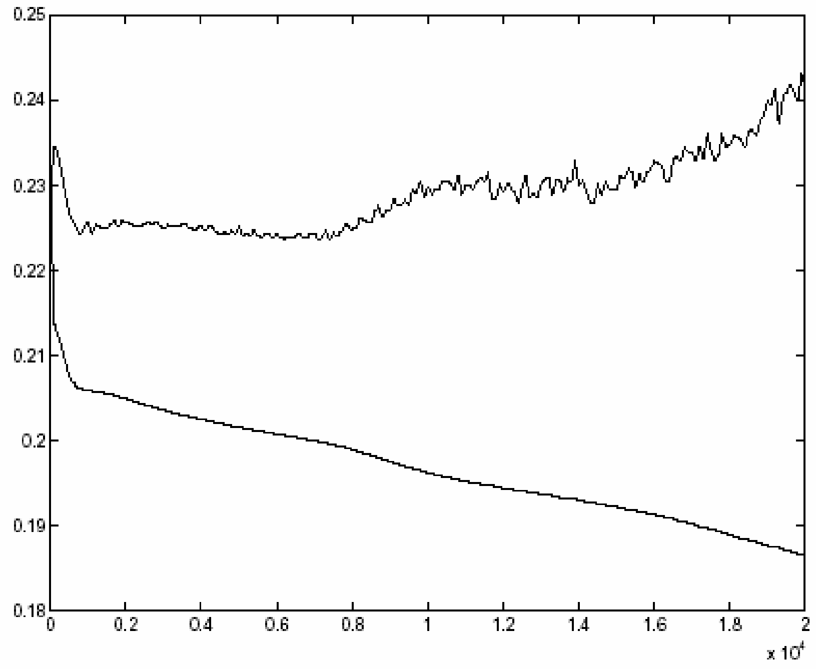
The lockback was set to 5 steps, and the data set was further divided into a training set and a validation set. The latter contained data from 1954 to 1979.



Using the `Backpropagation_v1.0` software, a 5-5-1 FFNN was trained to predict the rain fall in year i, given input from years i-1, i-2, ..., i-5.

The training and validation errors are shown in the figure. Note the clear case of overfitting, beginning around 7,000 epochs.

The network obtained after 7,000 epochs (i.e. at the minimum of the validation error) was used. The resulting predictions, however, were of the same quality (average error) as the simplest possible prediction, namely x(i) = x(i-1).

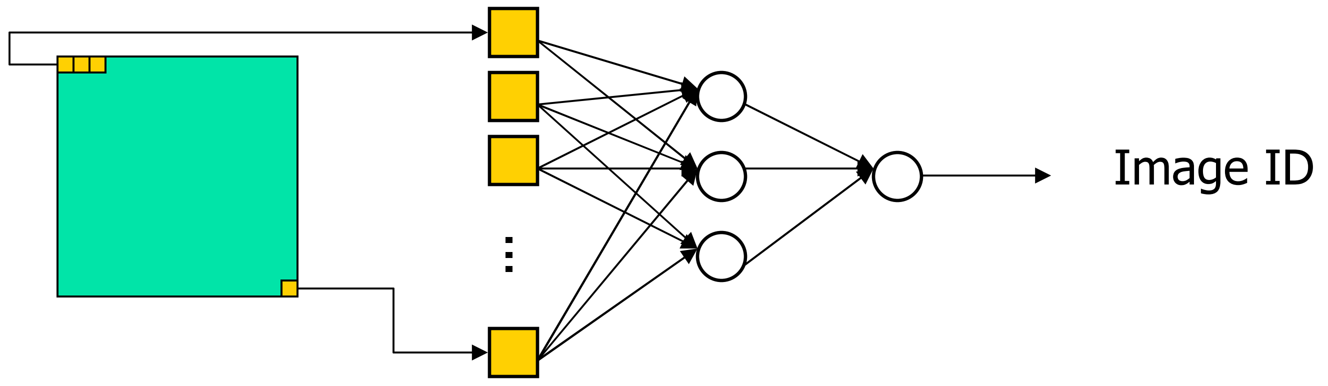


Always consider simple prediction methods (such as linear predictors); The results from an FFNN should be compared with the results from the simple predictors!

Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde

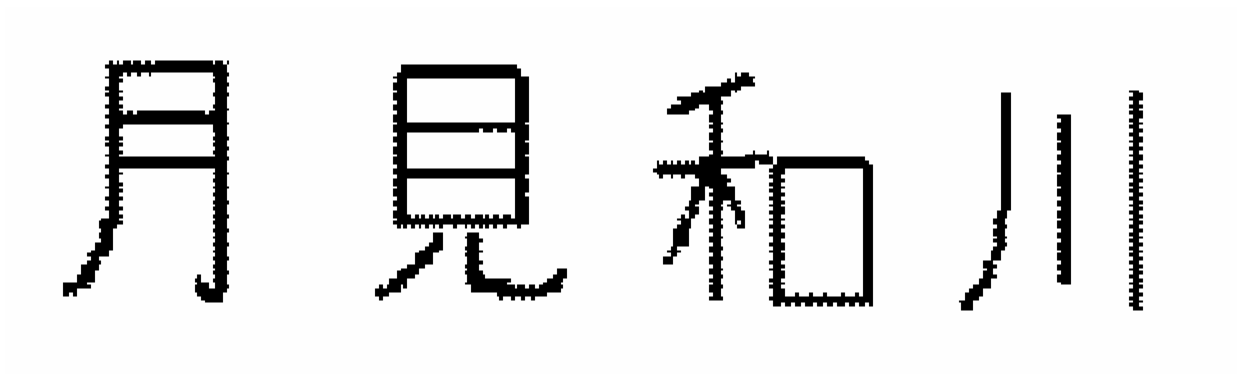
Image recognition

Automatic image recognition by neural networks is important in e.g. security systems, systems for automatic mail sorting, and for character recognition in e.g. PDAs.

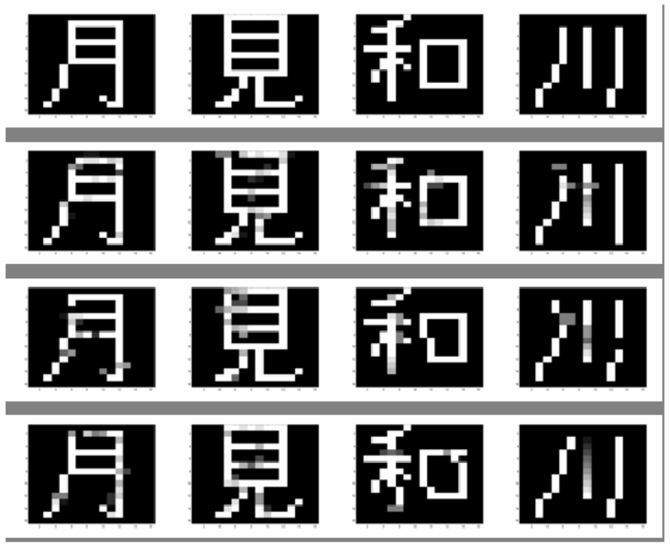


Example: automatic identification of Kanji signs

Some languages, e.g. Japanese and Chinese, are based on pictograms (called kanji, in the case of Japanese). Four examples of kanji signs are shown in the figure below. From left to right, the signs are *tsuki* (moon), *miru* (to see), *wa* (peace, harmony), and *kawa* (river).



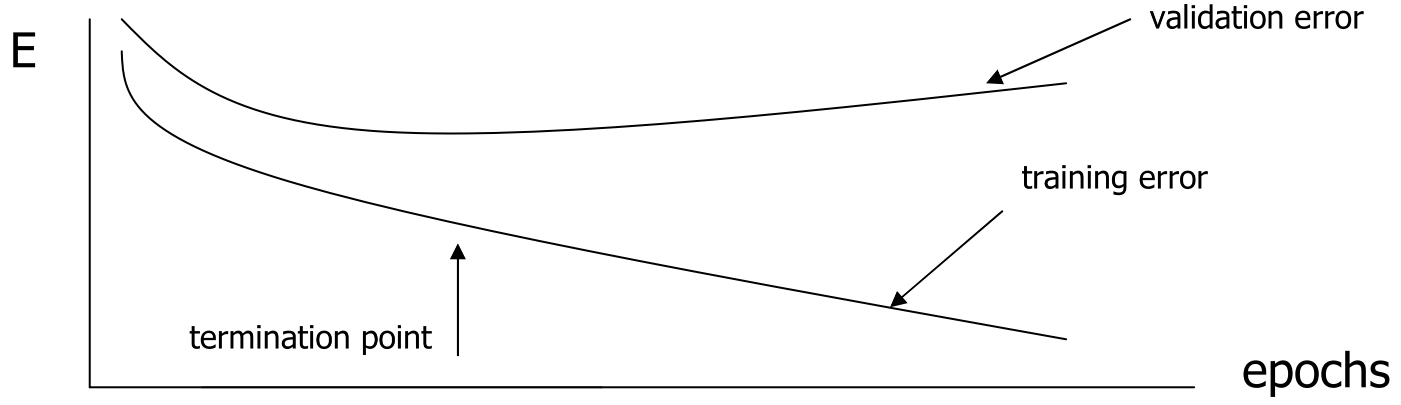
The signs on the previous slide were represented as 16x16 images, and the `Backpropagation_v1.0` program was used to train a 256-4-2 network to distinguish between these signs. A successful network was found quickly. The trained FFNN was not only able to distinguish between the four training images, but could also cope with noisy versions of the images (as shown in the figure on the right), in all cases producing correct output.



Thresholding was used, i.e. an output signa below a threshold T was set to 0, and and output signal above 1-T was set to 1. The four images were encoded in a binary fashion, i.e. 00 = first image, 01 = second image etc.

Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde

Overfitting: When backpropagation is applied to a training set, the error usually falls quickly at first, and then more slowly. When the RMS error falls below the noise level in the data set, the training should be terminated. However, in many cases the noise level of the data set is unknown, and there is a risk of training too much, so that the network is tuned to the noise (which does not increase the predictive capacity of the network). This is known as overfitting. In order to avoid overfitting, the data set should be split into one training set and one validation set. The training should be terminated before the error over the validation set starts increasing:



Mattias Wahde, PhD, associate professor, Chalmers University of Technology
e-mail: mattias.wahde@chalmers.se www: www.me.chalmers.se/~mwahde